package com.twitter.frigate.pushservice.predicate

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.\_

import com.twitter.frigate.common.candidate.MaxTweetAge

import com.twitter.frigate.common.candidate.TargetABDecider

import com.twitter.frigate.common.predicate.tweet.TweetAuthorPredicates

import com.twitter.frigate.common.predicate.\_

import com.twitter.frigate.common.rec\_types.RecTypes

import com.twitter.frigate.common.util.SnowflakeUtils

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.frigate.pushservice.params.PushParams

import com.twitter.frigate.pushservice.util.CandidateUtil

import com.twitter.frigate.thriftscala.ChannelName

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.gizmoduck.thriftscala.UserType

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.hermit.predicate.gizmoduck.\_

import com.twitter.hermit.predicate.socialgraph.Edge

import com.twitter.hermit.predicate.socialgraph.MultiEdge

import com.twitter.hermit.predicate.socialgraph.RelationEdge

import com.twitter.hermit.predicate.socialgraph.SocialGraphPredicate

import com.twitter.service.metastore.gen.thriftscala.Location

import com.twitter.socialgraph.thriftscala.RelationshipType

import com.twitter.stitch.tweetypie.TweetyPie.TweetyPieResult

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.Param

import com.twitter.util.Duration

import com.twitter.util.Future

object PredicatesForCandidate {

def oldTweetRecsPredicate(implicit stats: StatsReceiver): Predicate[

TweetCandidate with RecommendationType with TargetInfo[

TargetUser with TargetABDecider with MaxTweetAge

]

] = {

val name = "old\_tweet"

Predicate

.from[TweetCandidate with RecommendationType with TargetInfo[

TargetUser with TargetABDecider with MaxTweetAge

]] { candidate =>

{

val crt = candidate.commonRecType

val defaultAge = if (RecTypes.mrModelingBasedTypes.contains(crt)) {

candidate.target.params(PushFeatureSwitchParams.ModelingBasedCandidateMaxTweetAgeParam)

} else if (RecTypes.GeoPopTweetTypes.contains(crt)) {

candidate.target.params(PushFeatureSwitchParams.GeoPopTweetMaxAgeInHours)

} else if (RecTypes.simclusterBasedTweets.contains(crt)) {

candidate.target.params(

PushFeatureSwitchParams.SimclusterBasedCandidateMaxTweetAgeParam)

} else if (RecTypes.detopicTypes.contains(crt)) {

candidate.target.params(PushFeatureSwitchParams.DetopicBasedCandidateMaxTweetAgeParam)

} else if (RecTypes.f1FirstDegreeTypes.contains(crt)) {

candidate.target.params(PushFeatureSwitchParams.F1CandidateMaxTweetAgeParam)

} else if (crt == CommonRecommendationType.ExploreVideoTweet) {

candidate.target.params(PushFeatureSwitchParams.ExploreVideoTweetAgeParam)

} else

candidate.target.params(PushFeatureSwitchParams.MaxTweetAgeParam)

SnowflakeUtils.isRecent(candidate.tweetId, defaultAge)

}

}

.withStats(stats.scope(name))

.withName(name)

}

def tweetIsNotAreply(

implicit stats: StatsReceiver

): NamedPredicate[TweetCandidate with TweetDetails] = {

val name = "tweet\_candidate\_not\_a\_reply"

Predicate

.from[TweetCandidate with TweetDetails] { c =>

c.isReply match {

case Some(true) => false

case \_ => true

}

}

.withStats(stats.scope(name))

.withName(name)

}

/\*\*

\* Check if tweet contains any optouted free form interests.

\* Currently, we use it for media categories and semantic core

\* @param stats

\* @return

\*/

def noOptoutFreeFormInterestPredicate(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "free\_form\_interest\_opt\_out"

val tweetMediaAnnotationFeature =

"tweet.mediaunderstanding.tweet\_annotations.safe\_category\_probabilities"

val tweetSemanticCoreFeature =

"tweet.core.tweet.semantic\_core\_annotations"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

val withOptOutFreeFormInterestsCounter = stats.counter("with\_optout\_interests")

val withoutOptOutInterestsCounter = stats.counter("without\_optout\_interests")

val withOptOutFreeFormInterestsFromMediaAnnotationCounter =

stats.counter("with\_optout\_interests\_from\_media\_annotation")

val withOptOutFreeFormInterestsFromSemanticCoreCounter =

stats.counter("with\_optout\_interests\_from\_semantic\_core")

Predicate

.fromAsync { candidate: PushCandidate =>

val tweetSemanticCoreEntityIds = candidate.sparseBinaryFeatures

.getOrElse(tweetSemanticCoreFeature, Set.empty[String]).map { id =>

id.split('.')(2)

}.toSet

val tweetMediaAnnotationIds = candidate.sparseContinuousFeatures

.getOrElse(tweetMediaAnnotationFeature, Map.empty[String, Double]).keys.toSet

candidate.target.optOutFreeFormUserInterests.map {

case optOutUserInterests: Seq[String] =>

withOptOutFreeFormInterestsCounter.incr()

val optOutUserInterestsSet = optOutUserInterests.toSet

val mediaAnnoIntersect = optOutUserInterestsSet.intersect(tweetMediaAnnotationIds)

val semanticCoreIntersect = optOutUserInterestsSet.intersect(tweetSemanticCoreEntityIds)

if (!mediaAnnoIntersect.isEmpty) {

withOptOutFreeFormInterestsFromMediaAnnotationCounter.incr()

}

if (!semanticCoreIntersect.isEmpty) {

withOptOutFreeFormInterestsFromSemanticCoreCounter.incr()

}

semanticCoreIntersect.isEmpty && mediaAnnoIntersect.isEmpty

case \_ =>

withoutOptOutInterestsCounter.incr()

true

}

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def tweetCandidateWithLessThan2SocialContextsIsAReply(

implicit stats: StatsReceiver

): NamedPredicate[TweetCandidate with TweetDetails with SocialContextActions] = {

val name = "tweet\_candidate\_with\_less\_than\_2\_social\_contexts\_is\_not\_a\_reply"

Predicate

.from[TweetCandidate with TweetDetails with SocialContextActions] { cand =>

cand.isReply match {

case Some(true) if cand.socialContextTweetIds.size < 2 => false

case \_ => true

}

}

.withStats(stats.scope(name))

.withName(name)

}

def f1CandidateIsNotAReply(implicit stats: StatsReceiver): NamedPredicate[F1Candidate] = {

val name = "f1\_candidate\_is\_not\_a\_reply"

Predicate

.from[F1Candidate] { candidate =>

candidate.isReply match {

case Some(true) => false

case \_ => true

}

}

.withStats(stats.scope(name))

.withName(name)

}

def outOfNetworkTweetCandidateEnabledCrTag(

implicit stats: StatsReceiver

): NamedPredicate[OutOfNetworkTweetCandidate with TargetInfo[TargetUser with TargetABDecider]] = {

val name = "out\_of\_network\_tweet\_candidate\_enabled\_crtag"

val scopedStats = stats.scope(name)

Predicate

.from[OutOfNetworkTweetCandidate with TargetInfo[TargetUser with TargetABDecider]] { cand =>

val disabledCrTag = cand.target

.params(PushFeatureSwitchParams.OONCandidatesDisabledCrTagParam)

val candGeneratedByDisabledSignal = cand.tagsCR.exists { tagsCR =>

val tagsCRSet = tagsCR.map(\_.toString).toSet

tagsCRSet.nonEmpty && tagsCRSet.subsetOf(disabledCrTag.toSet)

}

if (candGeneratedByDisabledSignal) {

cand.tagsCR.getOrElse(Nil).foreach(tag => scopedStats.counter(tag.toString).incr())

false

} else true

}

.withStats(scopedStats)

.withName(name)

}

def outOfNetworkTweetCandidateEnabledCrtGroup(

implicit stats: StatsReceiver

): NamedPredicate[OutOfNetworkTweetCandidate with TargetInfo[TargetUser with TargetABDecider]] = {

val name = "out\_of\_network\_tweet\_candidate\_enabled\_crt\_group"

val scopedStats = stats.scope(name)

Predicate

.from[OutOfNetworkTweetCandidate with TargetInfo[TargetUser with TargetABDecider]] { cand =>

val disabledCrtGroup = cand.target

.params(PushFeatureSwitchParams.OONCandidatesDisabledCrtGroupParam)

val crtGroup = CandidateUtil.getCrtGroup(cand.commonRecType)

val candGeneratedByDisabledCrt = disabledCrtGroup.contains(crtGroup)

if (candGeneratedByDisabledCrt) {

scopedStats.counter("filter\_" + crtGroup.toString).incr()

false

} else true

}

.withStats(scopedStats)

.withName(name)

}

def outOfNetworkTweetCandidateIsNotAReply(

implicit stats: StatsReceiver

): NamedPredicate[OutOfNetworkTweetCandidate] = {

val name = "out\_of\_network\_tweet\_candidate\_is\_not\_a\_reply"

Predicate

.from[OutOfNetworkTweetCandidate] { cand =>

cand.isReply match {

case Some(true) => false

case \_ => true

}

}

.withStats(stats.scope(name))

.withName(name)

}

def recommendedTweetIsAuthoredBySelf(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate] =

Predicate

.from[PushCandidate] {

case tweetCandidate: PushCandidate with TweetDetails =>

tweetCandidate.authorId match {

case Some(authorId) => authorId != tweetCandidate.target.targetId

case None => true

}

case \_ =>

true

}

.withStats(statsReceiver.scope("predicate\_self\_author"))

.withName("self\_author")

def authorInSocialContext(implicit statsReceiver: StatsReceiver): NamedPredicate[PushCandidate] =

Predicate

.from[PushCandidate] {

case tweetCandidate: PushCandidate with TweetDetails with SocialContextActions =>

tweetCandidate.authorId match {

case Some(authorId) =>

!tweetCandidate.socialContextUserIds.contains(authorId)

case None => true

}

case \_ => true

}

.withStats(statsReceiver.scope("predicate\_author\_social\_context"))

.withName("author\_social\_context")

def selfInSocialContext(implicit statsReceiver: StatsReceiver): NamedPredicate[PushCandidate] = {

val name = "self\_social\_context"

Predicate

.from[PushCandidate] {

case candidate: PushCandidate with SocialContextActions =>

!candidate.socialContextUserIds.contains(candidate.target.targetId)

case \_ =>

true

}

.withStats(statsReceiver.scope(s"${name}\_predicate"))

.withName(name)

}

def minSocialContext(

threshold: Int

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with SocialContextActions] = {

Predicate

.from { candidate: PushCandidate with SocialContextActions =>

candidate.socialContextUserIds.size >= threshold

}

.withStats(statsReceiver.scope("predicate\_min\_social\_context"))

.withName("min\_social\_context")

}

private def anyWithheldContent(

userStore: ReadableStore[Long, User],

userCountryStore: ReadableStore[Long, Location]

)(

implicit statsReceiver: StatsReceiver

): Predicate[TargetRecUser] =

GizmoduckUserPredicate.withheldContentPredicate(

userStore = userStore,

userCountryStore = userCountryStore,

statsReceiver = statsReceiver,

checkAllCountries = true

)

def targetUserExists(implicit statsReceiver: StatsReceiver): NamedPredicate[PushCandidate] = {

TargetUserPredicates

.targetUserExists()(statsReceiver)

.flatContraMap { candidate: PushCandidate => Future.value(candidate.target) }

.withName("target\_user\_exists")

}

def secondaryDormantAccountPredicate(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "secondary\_dormant\_account"

TargetUserPredicates

.secondaryDormantAccountPredicate()(statsReceiver)

.on { candidate: PushCandidate => candidate.target }

.withStats(statsReceiver.scope(s"predicate\_$name"))

.withName(name)

}

def socialContextBeingFollowed(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with SocialContextActions] =

SocialGraphPredicate

.allRelationEdgesExist(edgeStore, RelationshipType.Following)

.on { candidate: PushCandidate with SocialContextActions =>

candidate.socialContextUserIds.map { u => Edge(candidate.target.targetId, u) }

}

.withStats(statsReceiver.scope("predicate\_social\_context\_being\_followed"))

.withName("social\_context\_being\_followed")

private def edgeFromCandidate(candidate: PushCandidate with TweetAuthor): Option[Edge] = {

candidate.authorId map { authorId => Edge(candidate.target.targetId, authorId) }

}

def authorNotBeingDeviceFollowed(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor] = {

SocialGraphPredicate

.relationExists(edgeStore, RelationshipType.DeviceFollowing)

.optionalOn(

edgeFromCandidate,

missingResult = false

)

.flip

.withStats(statsReceiver.scope("predicate\_author\_not\_device\_followed"))

.withName("author\_not\_device\_followed")

}

def authorBeingFollowed(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor] = {

SocialGraphPredicate

.relationExists(edgeStore, RelationshipType.Following)

.optionalOn(

edgeFromCandidate,

missingResult = false

)

.withStats(statsReceiver.scope("predicate\_author\_being\_followed"))

.withName("author\_being\_followed")

}

def authorNotBeingFollowed(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor] = {

SocialGraphPredicate

.relationExists(edgeStore, RelationshipType.Following)

.optionalOn(

edgeFromCandidate,

missingResult = false

)

.flip

.withStats(statsReceiver.scope("predicate\_author\_not\_being\_followed"))

.withName("author\_not\_being\_followed")

}

def recommendedTweetAuthorAcceptableToTargetUser(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor] = {

val name = "recommended\_tweet\_author\_acceptable\_to\_target\_user"

SocialGraphPredicate

.anyRelationExists(

edgeStore,

Set(

RelationshipType.Blocking,

RelationshipType.BlockedBy,

RelationshipType.HideRecommendations,

RelationshipType.Muting

)

)

.flip

.optionalOn(

edgeFromCandidate,

missingResult = false

)

.withStats(statsReceiver.scope(s"predicate\_$name"))

.withName(name)

}

def relationNotExistsPredicate(

edgeStore: ReadableStore[RelationEdge, Boolean],

relations: Set[RelationshipType]

): Predicate[(Long, Iterable[Long])] =

SocialGraphPredicate

.anyRelationExistsForMultiEdge(

edgeStore,

relations

)

.flip

.on {

case (targetUserId, userIds) =>

MultiEdge(targetUserId, userIds.toSet)

}

def blocking(edgeStore: ReadableStore[RelationEdge, Boolean]): Predicate[(Long, Iterable[Long])] =

relationNotExistsPredicate(

edgeStore,

Set(RelationshipType.BlockedBy, RelationshipType.Blocking)

)

def blockingOrMuting(

edgeStore: ReadableStore[RelationEdge, Boolean]

): Predicate[(Long, Iterable[Long])] =

relationNotExistsPredicate(

edgeStore,

Set(RelationshipType.BlockedBy, RelationshipType.Blocking, RelationshipType.Muting)

)

def socialContextNotRetweetFollowing(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with SocialContextActions] = {

val name = "social\_context\_not\_retweet\_following"

relationNotExistsPredicate(edgeStore, Set(RelationshipType.NotRetweetFollowing))

.optionalOn[PushCandidate with SocialContextActions](

{

case candidate: PushCandidate with SocialContextActions

if RecTypes.isTweetRetweetType(candidate.commonRecType) =>

Some((candidate.target.targetId, candidate.socialContextUserIds))

case \_ =>

None

},

missingResult = true

)

.withStats(statsReceiver.scope(s"predicate\_$name"))

.withName(name)

}

def socialContextBlockingOrMuting(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with SocialContextActions] =

blockingOrMuting(edgeStore)

.on { candidate: PushCandidate with SocialContextActions =>

(candidate.target.targetId, candidate.socialContextUserIds)

}

.withStats(statsReceiver.scope("predicate\_social\_context\_blocking\_or\_muting"))

.withName("social\_context\_blocking\_or\_muting")

/\*\*

\* Use hyrated Tweet object for F1 Protected experiment for checking null cast as Tweetypie hydration

\* fails for protected Authors without passing in Target id. We do this specifically for

\* F1 Protected Tweet Experiment in Earlybird Adaptor.

\* For rest of the traffic refer to existing Nullcast Predicate

\*/

def nullCastF1ProtectedExperientPredicate(

tweetypieStore: ReadableStore[Long, TweetyPieResult]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate with TweetDetails] = {

val name = "f1\_exempted\_null\_cast\_tweet"

val f1NullCastCheckCounter = statsReceiver.scope(name).counter("f1\_null\_cast\_check")

Predicate

.fromAsync { tweetCandidate: PushCandidate with TweetCandidate with TweetDetails =>

if (RecTypes.f1FirstDegreeTypes(tweetCandidate.commonRecType) && tweetCandidate.target

.params(PushFeatureSwitchParams.EnableF1FromProtectedTweetAuthors)) {

f1NullCastCheckCounter.incr()

tweetCandidate.tweet match {

case Some(tweetObj) =>

baseNullCastTweet().apply(Seq(TweetyPieResult(tweetObj, None, None))).map(\_.head)

case \_ => Future.False

}

} else {

nullCastTweet(tweetypieStore).apply(Seq(tweetCandidate)).map(\_.head)

}

}

.withStats(statsReceiver.scope(s"predicate\_$name"))

.withName(name)

}

private def baseNullCastTweet(): Predicate[TweetyPieResult] =

Predicate.from { t: TweetyPieResult => !t.tweet.coreData.exists { cd => cd.nullcast } }

def nullCastTweet(

tweetyPieStore: ReadableStore[Long, TweetyPieResult]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetCandidate] = {

val name = "null\_cast\_tweet"

baseNullCastTweet()

.flatOptionContraMap[PushCandidate with TweetCandidate](

f = (tweetCandidate: PushCandidate

with TweetCandidate) => tweetyPieStore.get(tweetCandidate.tweetId),

missingResult = false

)

.withStats(statsReceiver.scope(s"predicate\_$name"))

.withName(name)

}

/\*\*

\* Use the predicate except fn is true.

\*/

def exceptedPredicate[T <: PushCandidate](

name: String,

fn: T => Future[Boolean],

predicate: Predicate[T]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

Predicate

.fromAsync { e: T => fn(e) }

.or(predicate)

.withStats(statsReceiver.scope(name))

.withName(name)

}

/\*\*

\*

\* @param edgeStore [[ReadableStore[RelationEdge, Boolean]]]

\* @return - allow only out-network tweets if in-network tweets are disabled

\*/

def disableInNetworkTweetPredicate(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor] = {

val name = "disable\_in\_network\_tweet"

Predicate

.fromAsync { candidate: PushCandidate with TweetAuthor =>

if (candidate.target.params(PushParams.DisableInNetworkTweetCandidatesParam)) {

authorNotBeingFollowed(edgeStore)

.apply(Seq(candidate))

.map(\_.head)

} else Future.True

}.withStats(statsReceiver.scope(name))

.withName(name)

}

/\*\*

\*

\* @param edgeStore [[ReadableStore[RelationEdge, Boolean]]]

\* @return - allow only in-network tweets if out-network tweets are disabled

\*/

def disableOutNetworkTweetPredicate(

edgeStore: ReadableStore[RelationEdge, Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with TweetAuthor] = {

val name = "disable\_out\_network\_tweet"

Predicate

.fromAsync { candidate: PushCandidate with TweetAuthor =>

if (candidate.target.params(PushFeatureSwitchParams.DisableOutNetworkTweetCandidatesFS)) {

authorBeingFollowed(edgeStore)

.apply(Seq(candidate))

.map(\_.head)

} else Future.True

}.withStats(statsReceiver.scope(name))

.withName(name)

}

def alwaysTruePredicate: NamedPredicate[PushCandidate] = {

Predicate

.all[PushCandidate]

.withName("predicate\_AlwaysTrue")

}

def alwaysTruePushCandidatePredicate: NamedPredicate[PushCandidate] = {

Predicate

.all[PushCandidate]

.withName("predicate\_AlwaysTrue")

}

def alwaysFalsePredicate(implicit statsReceiver: StatsReceiver): NamedPredicate[PushCandidate] = {

val name = "predicate\_AlwaysFalse"

val scopedStatsReceiver = statsReceiver.scope(name)

Predicate

.from { candidate: PushCandidate => false }

.withStats(scopedStatsReceiver)

.withName(name)

}

def accountCountryPredicate(

allowedCountries: Set[String]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "AccountCountryPredicate"

val stats = statsReceiver.scope(name)

AccountCountryPredicate(allowedCountries)

.on { candidate: PushCandidate => candidate.target }

.withStats(stats)

.withName(name)

}

def paramPredicate[T <: PushCandidate](

param: Param[Boolean]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[T] = {

val name = param.getClass.getSimpleName.stripSuffix("$")

TargetPredicates

.paramPredicate(param)

.on { candidate: PushCandidate => candidate.target }

.withStats(statsReceiver.scope(s"param\_${name}\_controlled\_predicate"))

.withName(s"param\_${name}\_controlled\_predicate")

}

def isDeviceEligibleForNewsOrSports(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "is\_device\_eligible\_for\_news\_or\_sports"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: PushCandidate =>

candidate.target.deviceInfo.map(\_.exists(\_.isNewsEligible))

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def isDeviceEligibleForCreatorPush(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "is\_device\_eligible\_for\_creator\_push"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: PushCandidate =>

candidate.target.deviceInfo.map(\_.exists(settings =>

settings.isNewsEligible || settings.isRecommendationsEligible))

}

.withStats(scopedStatsReceiver)

.withName(name)

}

/\*\*

\* Like [[TargetUserPredicates.homeTimelineFatigue()]] but for candidate.

\*/

def htlFatiguePredicate(

fatigueDuration: Param[Duration]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "htl\_fatigue"

Predicate

.fromAsync { candidate: PushCandidate =>

val \_fatigueDuration = candidate.target.params(fatigueDuration)

TargetUserPredicates

.homeTimelineFatigue(

fatigueDuration = \_fatigueDuration

).apply(Seq(candidate.target)).map(\_.head)

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

def mrWebHoldbackPredicate(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "mr\_web\_holdback\_for\_candidate"

val scopedStats = stats.scope(name)

PredicatesForCandidate.exludeCrtFromPushHoldback

.or(

TargetPredicates

.webNotifsHoldback()

.on { candidate: PushCandidate => candidate.target }

)

.withStats(scopedStats)

.withName(name)

}

def candidateEnabledForEmailPredicate(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "candidates\_enabled\_for\_email"

Predicate

.from { candidate: PushCandidate =>

if (candidate.target.isEmailUser)

candidate.isInstanceOf[TweetCandidate with TweetAuthor with RecommendationType]

else true

}

.withStats(stats.scope(name))

.withName(name)

}

def protectedTweetF1ExemptPredicate[

T <: TargetUser with TargetABDecider,

Cand <: TweetCandidate with TweetAuthorDetails with TargetInfo[T]

](

implicit stats: StatsReceiver

): NamedPredicate[

TweetCandidate with TweetAuthorDetails with TargetInfo[

TargetUser with TargetABDecider

]

] = {

val name = "f1\_exempt\_tweet\_author\_protected"

val skipForProtectedAuthorScope = stats.scope(name).scope("skip\_protected\_author\_for\_f1")

val authorIsProtectedCounter = skipForProtectedAuthorScope.counter("author\_protected\_true")

val authorIsNotProtectedCounter = skipForProtectedAuthorScope.counter("author\_protected\_false")

val authorNotFoundCounter = stats.scope(name).counter("author\_not\_found")

Predicate

.fromAsync[TweetCandidate with TweetAuthorDetails with TargetInfo[

TargetUser with TargetABDecider

]] {

case candidate: F1Candidate

if candidate.target.params(PushFeatureSwitchParams.EnableF1FromProtectedTweetAuthors) =>

candidate.tweetAuthor.foreach {

case Some(author) =>

if (GizmoduckUserPredicate.isProtected(author)) {

authorIsProtectedCounter.incr()

} else authorIsNotProtectedCounter.incr()

case \_ => authorNotFoundCounter.incr()

}

Future.True

case cand =>

TweetAuthorPredicates.recTweetAuthorProtected.apply(Seq(cand)).map(\_.head)

}

.withStats(stats.scope(name))

.withName(name)

}

/\*\*

\* filter a notification if user has already received ANY prior notification about the space id

\* @param stats

\* @return

\*/

def duplicateSpacesPredicate(

implicit stats: StatsReceiver

): NamedPredicate[Space with PushCandidate] = {

val name = "duplicate\_spaces\_predicate"

Predicate

.fromAsync { c: Space with PushCandidate =>

c.target.pushRecItems.map { pushRecItems =>

!pushRecItems.spaceIds.contains(c.spaceId)

}

}

.withStats(stats.scope(name))

.withName(name)

}

def filterOONCandidatePredicate(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "filter\_oon\_candidate"

Predicate

.fromAsync[PushCandidate] { cand =>

val crt = cand.commonRecType

val isOONCandidate =

RecTypes.isOutOfNetworkTweetRecType(crt) || RecTypes.outOfNetworkTopicTweetTypes

.contains(crt) || RecTypes.isOutOfNetworkSpaceType(crt) || RecTypes.userTypes.contains(

crt)

if (isOONCandidate) {

cand.target.notificationsFromOnlyPeopleIFollow.map { inNetworkOnly =>

if (inNetworkOnly) {

stats.scope(name, crt.toString).counter("inNetworkOnlyOn").incr()

} else {

stats.scope(name, crt.toString).counter("inNetworkOnlyOff").incr()

}

!(inNetworkOnly && cand.target.params(

PushFeatureSwitchParams.EnableOONFilteringBasedOnUserSettings))

}

} else Future.True

}

.withStats(stats.scope(name))

.withName(name)

}

def exludeCrtFromPushHoldback(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = Predicate

.from { candidate: PushCandidate =>

val crtName = candidate.commonRecType.name

val target = candidate.target

target

.params(PushFeatureSwitchParams.CommonRecommendationTypeDenyListPushHoldbacks)

.exists(crtName.equalsIgnoreCase)

}

.withStats(stats.scope("exclude\_crt\_from\_push\_holdbacks"))

def enableSendHandlerCandidates(implicit stats: StatsReceiver): NamedPredicate[PushCandidate] = {

val name = "sendhandler\_enable\_push\_recommendations"

PredicatesForCandidate.exludeCrtFromPushHoldback

.or(PredicatesForCandidate.paramPredicate(

PushFeatureSwitchParams.EnablePushRecommendationsParam))

.withStats(stats.scope(name))

.withName(name)

}

def openAppExperimentUserCandidateAllowList(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "open\_app\_experiment\_user\_candidate\_allow\_list"

Predicate

.fromAsync { candidate: PushCandidate =>

val target = candidate.target

Future.join(target.isOpenAppExperimentUser, target.targetUser).map {

case (isOpenAppUser, targetUser) =>

val shouldLimitOpenAppCrts =

isOpenAppUser || targetUser.exists(\_.userType == UserType.Soft)

if (shouldLimitOpenAppCrts) {

val listOfAllowedCrt = target

.params(PushFeatureSwitchParams.ListOfCrtsForOpenApp)

.flatMap(CommonRecommendationType.valueOf)

listOfAllowedCrt.contains(candidate.commonRecType)

} else true

}

}.withStats(stats.scope(name))

.withName(name)

}

def isTargetBlueVerified(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "is\_target\_already\_blue\_verified"

Predicate

.fromAsync { candidate: PushCandidate =>

val target = candidate.target

target.isBlueVerified.map(\_.getOrElse(false))

}.withStats(stats.scope(name))

.withName(name)

}

def isTargetLegacyVerified(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "is\_target\_already\_legacy\_verified"

Predicate

.fromAsync { candidate: PushCandidate =>

val target = candidate.target

target.isVerified.map(\_.getOrElse(false))

}.withStats(stats.scope(name))

.withName(name)

}

def isTargetSuperFollowCreator(implicit stats: StatsReceiver): NamedPredicate[PushCandidate] = {

val name = "is\_target\_already\_super\_follow\_creator"

Predicate

.fromAsync { candidate: PushCandidate =>

val target = candidate.target

target.isSuperFollowCreator.map(

\_.getOrElse(false)

)

}.withStats(stats.scope(name))

.withName(name)

}

def isChannelValidPredicate(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "is\_channel\_valid"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: PushCandidate =>

candidate

.getChannels().map(channels =>

!(channels.toSet.size == 1 && channels.head == ChannelName.None))

}

.withStats(scopedStatsReceiver)

.withName(name)

}

}